Preliminary Concepts of Algorithm and Data Structure

by Xingyu Yang

Contents

[1] This is optional, only for math lovers! Recommended for those who know function, set, and analysis well.

§1. Defining Algorithm and Data Structure

It is always important to understand the basic concepts before we start to learn about the more advanced topics. In this part, we will first introduce the concept of algorithm and data structure, and then we will discuss the complexity of algorithms and data structures.

§1.1. Algorithm

What is algorithm? The answer may varie even among the computer scientists, since definition can be quite board as long as they fit into most part of the big picture. However, we can define algorithm, in general sense, as a set of instructions that are used to solve a problem or perform a task. In other words, an algorithm is a step-by-step procedure that is used to solve a problem or perform a task. Formally, in rigorous mathematical language, we have the following definition.

Definition 1.1.1 (Algorithm).
Algorithm is a **finite** sequence of **well-defined**, **computer-implementable** instructions, typically to solve a class of problems or to perform a computation.

Formally, an algorithm is a function $A : \Sigma^* \to \Sigma^*$ that is computable by a Turing machine M, where Σ is a finite alphabet, and Σ^* represents the set of all finite strings over Σ. The function A must satisfy the following properties:

1. **Finiteness**: M must halt for all valid inputs within a finite number of steps.
2. **Well-defined**: For every input $x \in \Sigma^*$, the output $A(x)$ is uniquely determined.
3. **Effectiveness**: The function A can be implemented using a finite set of instructions that can be executed mechanically (e.g., by a Turing machine or an equivalent model of computation).

To elaborate, an algorithm must has a stop condition, which means it must halt for all valid inputs within a finite number of steps, otherwise, it will be an infinite loop. Moreover, the output of the algorithm must be uniquely determined for every input, which means the algorithm must be deterministic. Finally, the algorithm must be implementable using a finite set of instructions that can be executed mechanically, such as a Turing machine or an equivalent model of computation (lambda calculus or recursive function).

Example (Macthing String with Turing Machine[2]).
Consider an example of a trivial algorithm that determine whether a given string is consist of only character '0'.

This algorithm is runed by a simple Turing machine that recognizes the language

$$L = \{0^n \mid n \geq 0\}.$$

This means the machine accepts any string containing only the character `0` (and no any other symbols).

[2]Click here in case that you do not know what is a "Turing Machine".

To make such a machine, we simply need to set up the following rules. First, we determine the **possible states** of the machine:

1. q_0: the initial state
2. q_{accept}: the accepting state
3. q_{reject}: the rejecting state

Then, we define the **transition rules** for the machine:

1. If we are in state q_0 and we see a `0`, we stay in state q_0 and move right.
2. If we are in state q_0 and we see a blank symbol, this means that the end of string has been reached, and we move to state q_{accept} and move right, return accept.
3. If we are in state q_0 and we see any other symbol, we move to state q_{reject} and move right, return reject.

Throughout the excecution, we keep track of the current state, the current symbol, and the position of the head on the tape. At the beginning, the tape is filled with blanks, and the head is positioned at the beginning of the input string. The machine will keep running until it reaches the accepting or rejecting state. On top of that, the way of transition is **deterministic** by the rules we defined.

The following Python code shows the implementation of the Turing machine that recognizes the language $L = \{0^n \mid n \geq 0\}$. This is optional for those who want to take a more challenging approach to understand the concept of algorithm and Turing machine. While for most, you may only need to understand how a turing machine embodies the characteristics of an algorithm without knowing pricise implementation.

```python
class TuringMachine:
    def __init__(self, transitions, start_state, accept_state, reject_state):
        """
        transitions: dict with keys  -> (state, symbol)
                     and values -> (new_state, new_symbol, direction)
        """
        self.transitions = transitions
        self.state = start_state
        self.accept_state = accept_state
        self.reject_state = reject_state
        self.tape = ["_"] * 100  # Blank tape (bigger just for safety)
        self.head = 50           # Start in the middle

    def run(self, input_string):
        """Loads the input and simulates the Turing Machine."""
        # 1. Load the input onto the tape
        for i, symbol in enumerate(input_string):
            self.tape[self.head + i] = symbol

        # 2. Keep running until we accept or reject
        while self.state not in (self.accept_state, self.reject_state):
            current_symbol = self.tape[self.head]

            if (self.state, current_symbol) in self.transitions:
```

```
                new_state, new_symbol, direction = self.transitions[(self.state, current_symbol)]

                # Write the new symbol
                self.tape[self.head] = new_symbol
                # Move to the new state
                self.state = new_state
                # Move head left or right
                if direction == "R":
                    self.head += 1
                else:  # "L"
                    self.head -= 1
            else:
                # No valid transition => reject
                self.state = self.reject_state

        # 3. Check whether we've accepted or rejected
        return "Accepted" if self.state == self.accept_state else "Rejected"

# ---------------------------------------------------------
# Transitions for L = { all strings of 0's }
# ---------------------------------------------------------
# Explanation:
# (q0, '0') -> (q0, '0', R): Keep scanning right if we see a '0'
# (q0, '_') -> (q_accept, '_', R): If we see blank, we accept
#              (means only zeros so far, done reading)
# Anything else => reject (by default: no transition => reject)

transitions = {
    ("q0", "0"): ("q0", "0", "R"),      # see a zero, stay in q0, move right
    ("q0", "_"): ("q_accept", "_", "R") # see a blank, accept
}
def create_tm() -> TuringMachine:
    return TuringMachine(
        transitions=transitions,
        start_state="q0",
        accept_state="q_accept",
        reject_state="q_reject"
    )

def reset() -> None:
    # using global variable only for convenience here, not a good practice
    global tm
    tm = create_tm()

# Test Cases
```

```python
tm = create_tm()
print("Input ''        ->", tm.run(""))       # "" (empty string) => Accepted
reset()
print("Input '0'       ->", tm.run("0"))      # "0" => Accepted
reset()
print("Input '000'     ->", tm.run("000"))    # "000" => Accepted
reset()
print("Input '001'     ->", tm.run("001"))    # Contains a '1' => Rejected
reset()
print("Input '111'     ->", tm.run("111"))    # All ones => Rejected
```

This interesting example has brought us to the intersection of computability theory and algorithm theory. The concept of an algorithm is not just about the code we write but also about the theoretical models of computation, such as the Turing machine, which forms the foundation of modern computer science. Moreover, we can see that the computational model intertwines with the concept of an algorithm and its three defining characteristics. An intriguing fact is that whether a problem is computable remains an open question in computer science—there exists no algorithm that can determine whether the execution of a given algorithm (modeled as a Turing machine) will halt. This is known as the Halting Problem, one of the most famous undecidable problems in computer science.

§1.2. Data Structure

All Comp Sci students are familiar with the phrase "Algorithm and Data Structure", and many people may get confused when these two concepts are juxtaposed. This section focuses on the definition of Data Structure, its history, purpose, as well as how it is related to algorithm.

§1.2.1. Definition of Data Structure

Before diving deep into the practical applications and historical evolution of data structures, it is essential to establish a clear and comprehensive definition. Data structures are more than just means of arranging data in computing; they are the backbone of effective algorithm design and the efficient execution of software. Here are several definitions that illuminate different facets of data structures:

> **Definition 1.2.1.1** (Data Structure).
> Data Structure is a **collection of data values**, the relationships among them, and the functions or operations that can be applied to the data. It can be any format regardless of tangible or intangible, and it can be used to store, organize, and manage data typically in a computer, but can also used in real life scenarios in a broader sense.

While this is a general definition to Data Structure, we can also extend it in different context.

- **Technical Perspective**: Data structures are mechanisms for organizing data in a computer's memory or storage in such a way that it can be accessed and modified efficiently. Common examples include arrays, linked lists, stacks, queues, trees, and hash tables.

- **Algorithmic Utility**: From the standpoint of algorithms, data structures are indispensable tools that facilitate data management and manipulation, allowing for robust data processing. This includes operations like searching, sorting, and maintaining large datasets.
- **Programming Integration**: In the realm of programming, data structures are critical constructs that enable the storage and organization of data so that it can be utilized effectively in software development.
- **Abstract Data Types (ADT)**: Often, data structures are implemented as specific instances of abstract data types that define the model by the operations it supports, such as adding, removing, or finding data in a structured way.
- **Theoretical and Mathematical Models**: Theoretically, data structures can be viewed as mathematical models that describe the logical relationship between individual elements of data. This perspective is vital for theoretical computer science where data structures are studied abstractly.

This variety of definitions helps highlight the versatility and centrality of data structures in computer science. They underscore the importance of choosing the right data structure for a particular problem, as it can significantly impact the efficiency and clarity of the solution.

§1.2.2. Ancient Data Structures

Modern Data Structures are initiated and developed around 1950s, however, Data Structure spans a history of more than thousands of years, throughout our civilisation history. The concept of data structures can be traced back to ancient times when humans began to record information and organize it in various forms. Data structure is not cling to the computer science, but it is a fundamental concept that has been utilized by humans for centuries. As long as there is information to be stored, retrieved, and processed, there exists a need for data structures.

While modern data structures as we understand them today began to take shape around the 1950s with the advent of computer science, the concept of organizing information systematically goes much further back in human history. Here are some significant examples of ancient data structures.

§1.2.2.1. Tally Sticks

Figure 1: A tally stick used for record-keeping.

Tally sticks, one of the earliest tools for recording and documenting information, have a history spanning thousands of years. Initially appearing as carved animal bones during the Upper Paleolithic, tally sticks evolved into sophisticated devices for counting, bookkeeping, and financial transactions. By the medieval period, they became central to economic systems, particularly in England under King Henry I, who introduced the tally stick system around 1100 AD. Made from polished wood such as hazel or willow, these sticks featured notches of varying sizes to denote specific values. Their unique design involved splitting the stick lengthwise, with each party to a transaction retaining one half, ensuring security and authenticity through the matching of the two halves. Beyond record-keeping, tally sticks were used as currency and instruments of credit, circulating in secondary markets as 'wooden money.' Their durability and resistance to forgery made them indispensable in financial and administrative contexts until their gradual decline in the 19th century.

§1.2.2.2. Clay Tablets

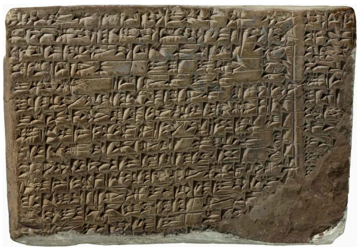


Figure 2: A clay tablet inscribed with cuneiform writing.

Clay tablets were one of the earliest writing mediums, widely used in ancient Mesopotamia and neighboring regions from the 5th millennium BCE through the Iron Age. Made from soft clay, these tablets were inscribed with cuneiform characters using a stylus, often crafted from reed, and then either sun-dried or kiln-baked for preservation. They varied in size and shape, ranging from small rectangular pieces to circular forms, and were used to record a vast array of information, including legal codes, trade transactions, myths, administrative records, and epic literature like the Epic of Gilgamesh. Their durability allowed many to survive for thousands of years, providing invaluable insights into ancient civilizations. Clay tablets also played a crucial role in the development of cuneiform writing and the establishment of early archives and libraries, making them a cornerstone of human history and a testament to early human ingenuity in preserving knowledge.

§1.2.2.3. Quipus

Figure 3: A quipu for recording information.

Quipus, or khipus, were sophisticated recording devices used by the Inca Empire and earlier Andean civilizations to store and communicate information. Comprising a primary cord with hanging strings of various colors and knots, quipus encoded data through a combination of knot types, positions, and string arrangements. This system, based on a decimal positional structure, allowed the Incas to record numerical information such as census data, tax obligations, and agricultural output. Beyond their practical use in administration, quipus may have also conveyed historical narratives and cultural knowledge. First emerging as early as 2600 BCE and flourishing during the Inca Empire (1438-1533 CE), quipus were vital for managing the vast Andean territory, especially in regions with diverse languages and cultures. Though many were destroyed after the Spanish conquest, surviving examples continue to intrigue researchers and symbolize the ingenuity of Andean civilizations. Nevertheless, the exact method of reading and interpreting quipus remains a subject of ongoing research and debate, as noone has been able to fully decipher their complex encoding system.

These ancient systems show that the fundamental concept of a data structure—to efficiently organize, store, and retrieve data—has been integral to human progress throughout history. Understanding these origins enriches our appreciation of modern data structures, highlighting a continuous thread of innovation and adaptation.

§1.2.3. Modern Data Structures

The term “data structure” as we know it today emerged in the mid-20th century with the rise of computer science as a distinct field of study. The development of data structures was driven by the need to manage and manipulate data efficiently in computer systems, leading to the creation of a wide range of structures tailored to specific computational tasks. Here are some key milestones in the evolution of modern data structures.We will give a brief introduction to some of the most fundamental data structures, which are served as cornnerstones for more complex data structures we will discuss later.

§1.2.3.1. Array: Primitive Linear Data Structures

We can start with the most basic data structures, which are the building blocks of more complex structures. We will first discuss linear data structures, which organize data in a sequential manner.

Definition 1.2.3.1.1 (Linear Data Structure).
A linear data structure is a collection of data elements arranged in a sequential order, where each element is connected to its previous or next element. Linear data structures are characterized by their simplicity and ease of traversal[3], making them essential components of more complex data structures

The simplest and most fundamental linear data structure is the array.

Definition 1.2.3.1.2 (Array).
Arrays are one of the simplest and most fundamental data structures, consisting of a collection of elements stored in contiguous memory locations. They provide efficient access to individual elements based on their index, making them ideal for tasks that require random access or sequential processing. Arrays are widely used in programming languages and form the basis for more complex data structures. As one of the earliest data structures, arrays are functionally limited:

- Fixed size: Arrays have a fixed size determined at creation, making it challenging to resize them dynamically.
- Homogeneous elements: Arrays store elements of the same data type, limiting their flexibility for heterogeneous data.
- Memory allocation: Arrays require contiguous memory allocation, which can be inefficient for large or variable-sized data.

Example.
We can define a fix-sized array to store the name of members in a family of three. The array is defined as `family = ["Alice", "Bob", "Charlie"]`. We can access the name of the first member by `family[0]`, the second member by `family[1]`, and the third member by `family[2]`.

The way we access data by index is known as **random access**, which is a key feature of arrays, and some other data structures we will discuss later. For your device to fetch data from an array, we must use a meta data called pointer, which is a memory address that points to the location where the first element of the array is stored. By using the pointer and the index,

[3]Process of moving through or a structure, in a methodical manner.

we can calculate the memory address of the element we want to access. For example, assume that the previous array `family` is stored in memory starting from address `0x1000`, and each element occupies 4 bytes. Then, the memory address of the first element `family[0]` is `0x1000`, the second element `family[1]` is `0x1004`, and the third element `family[2]` is `0x1008`. Whenever a random access is made, the device will calculate the memory address of the element by adding the index to the base address of the array. This is how random access works in many linear data structures.

Speak of the rigidness of arrays, we can also see that the fixed size of arrays can be a limitation in some cases. For example, if we want to store the names of a family, but we don't know how many members are there, we cannot use an array to store the names, or we want an array for all the names and age of the family members, which are of different data types, we cannot use an array to store them. These requires us to design more flexible data structures, which we will discuss later.

Despite this is a very simple data structure, it still takes some efforts to figure out the way to implement them. But for now, we only need to know what is it and how it works, we will discuss on the base of this knowledge to introduce the idea of algorithm complexity.

§1.2.3.2. Object-Oriented Design and Data Structures

We assume the readers understand the concept of object-oriented programming, which is a programming paradigm that uses objects and classes to design and implement software systems. Thus, we will not discuss the basic concept of OOP here. Instead, we will focus on how data structures are designed and implemented in an object-oriented manner. As we mentioned earlier, array, as a basic data structure, can be refined and extented to some more fancy data structures, such as linked list, stack, queue, etc. This is where the concept of object-oriented design comes in. We may use previously designed classes to build more complex data structures. Moreover, we will learn more advanced OOP concepts, such as abstract base case or ADT, abstract data type, which is a model for data types where the data is defined by its behavior (semantics) from the point of view of a user of the data, specifically in terms of possible values, possible operations on data of this type, and the behavior of these operations.

Still, taking array as an example. We have mentioned that array is a linear data structure, which encompasses many other linear data structures, such as linked list, stack, queue, etc. Therefore, naturally, the class `linear_data_structure` must be something very general, which can be used to define all the linear data structures. This means that, we will define all behaviors of this class, except those method or attributes that all, or at least most of the sub-linear structures have in common. Actually, we define something called abstract class.

> **Definition 1.2.3.2.1** (Abstract Class).
> Abstract class is a class that cannot be instantiated on its own and is designed to be subclassed by other classes. It may contain one or more abstract methods, which are methods that are declared but not implemented in the abstract class. Subclasses of the abstract class must implement the abstract methods to provide concrete functionality.

Since we use python as our primary language, we use python to demonstrate how to implement an abstract class using a trivial case. We will show a trivial case that' implement a class `Dog` from an abstract class `Animal`.

```python
from abc import ABC, abstractmethod
# abc is the python module that provides the abstract class
from typing import *
# typing is the python module that provides the type hinting, though it is not necessary, it is good practice to use it.

# Define the abstract class Animal, inherited from ABC, abstract base class
class Animal(ABC):
    def __init__(self, name: str, age: int, species: str):
    """
    Constructor for the Animal class. Unify the attributes of all animals.
    # However, note that the constructor can sometimes be abstract, which means that the constructor is not implemented in the abstract class, but it is implemented in the subclass, not like what we do here.
    """
        self.name = name
        self.age = age
        self.species = species

    @abstractmethod
    def make_sound(self) -> None:
        """
        Abstract method for making sound. This method must be implemented in the subclass. Because different animals make different sounds.
        """
        pass
        # pass is a keyword in python that does nothing, it is used to fill the
        # block of code that is required but we don't want to do anything. the #
        interpreter will ignore this line, and will not raise any error.

    # we ignore other methods here, since they just does not matter to the example.

    # Define the subclass Dog, inherited from Animal
    class Dog(Animal):
        def __init__(self, name: str, age: int):
            """
            Constructor for the Dog class. Inherited from the Animal class.
            """
            super().__init__(name, age, "Dog")
            # super() is a python function that returns a temporary object of the superclass that allows us to call the superclass's methods. If the constructor of abstract class is abstract, we need to define the constructor in the subclass instead.

        def make_sound(self) -> None:
            print("Woof! Woof!")

```

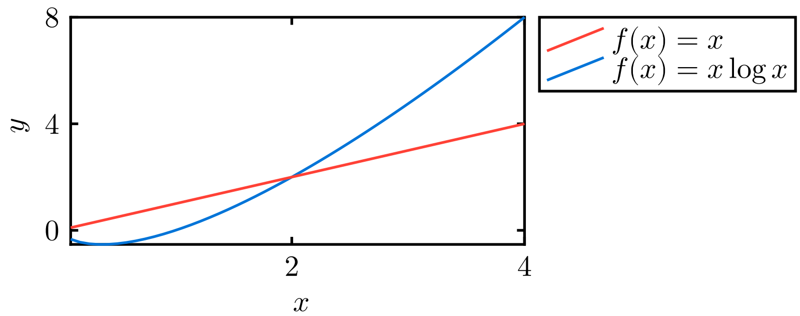
```python
# The other class Cat
class Cat(Animal):
    def __init__(self, name: str, age: int):
        """
        Constructor for the Cat class. Inherited from the Animal class.
        """
        super().__init__(name, age, "Cat")

    def make_sound(self) -> None:
        print("Meow! Meow!")
```

This example how abstract class and methods works in python. If we do not implement any one of the abstract methods in the subclass, the python interpreter will refuse to instantiate the subclass object and raise an error. Using abstract class and abstract methods is in most cases a good practice in OOP, since it can help us to design the class hierarchy more clearly and stipulate the behaviors of the subclasses.

sured in terms of the number of elementary operations performed by the algorithm, such as comparisons, assignments, and arithmetic operations.

Why does input size matter? We can illustrate using function graphs. can you write a typst definition environment on the definition of general algorithm time complexity without using too much formal definitions in math?



As we can see, the function $f(x) = x$ grows linearly with the input size x, while the function $f(x) = x \log x$ grows logarithmically with the input size x. It's obvious that they intersect at $x \approx 2$. Now we take the function value as algorithm complexity, and it can be interpreted that when input size is less than 2, the linear algorithm is more efficient, while when input size is larger than 2, the logarithmic algorithm is more efficient. This basic example shows why input size always matters in the analysis of algorithm complexity. Because we cannot find something like a one-size-fits-all algorithm that is always the most efficient, we also need to consider the input size and other limitations/constraintswhen we choose an algorithm to solve a problem.

§1.3. Time Complexity

The previous section gives the brief intro to algorithm complexity, and we will now discuss the time complexity in more detail. The time complexity of an algorithm is a measure of the amount of time it takes to run as a function of the input size. Well, it could be a bit vague, since some may believe that we really bechmark algorithm based on the actual time it takes to run, but it is not the case. The time complexity is usually measured in terms of the number of elementary operations performed by the algorithm, such as comparisons, assignments, and arithmetic operations. We take the assumption that the time taken for each elementary operation is constant, and all machines deal with the same elementary operations in the same amount of time[4].

Definition 1.3.1 (Time Complexity).
The **time complexity** describes how the running time of the algorithm grows as the size of the input increases. It can be expressed using different notations given varied purposes, which provides an upper bound, lower bound or average performance of the algorithm.

Mathematically, time complexity can be defined as a function $T : A \times x \to F(x)$, where A is the set of all algorithms[5], n is the input size,and $F(x)$ is the set of possible functions with x as input, normally, $F(x) = \{1, x^n, n^x, \log_n(x), \ln(x), x\log_n(x), x!\}$, where $n \in \mathbb{N}^+$, x is the input size of the algorithm. We have time complexity $T(a) = f(n)$ for $a \in A$ and $f(n) \in F(n)$ when the input size of a is n.

This could be a bit abstract, but this is not really the common time complexity definition we use in practice. In the real practice, we add another layer of abstraction to the time complexity, bechmarking algorithms in different aspects, such as the worst-case, best-case, and average-case time complexity. We will discuss them in the following sections.

Linear Search

```
for i = 0 to n - 1:
    if arr[i] == target then:
        return i
    end
end
return −1
```

Let's Consider the example of liearn search, a simple algorithm that searches for a target element in an array. The pseudocode of the linear search algorithm is shown along size. We simply designate a target element and traverse the array from the beginning to the end, comparing each element with the target. If the target is found, we return the index of the element; otherwise, we return -1 to indicate that the target is not in the array. Now we will analyze the time complexity of the linear search algorithm.

Example.
To figure out the time complexity of the linear search, we first need to determine, which metadata decides the input size of the algorithm. Obviously, in this algorithm we only traverse the array linearly, which means the input size of the algorithm is the length of the array. Therefore, we denote the input size as n, and the time complexity of the linear search algorithm is n, where n is the length of the array.

Now we look at a slightly different example, which is also related to Linear Search.

```python
def linear_search(arr: List[int], target: int) -> int:
    """
```

[4]But not literally the same, since the performance of different machine differs, however, the computing overheads are very tiny so that the gap is trivial and can be ignored.

[5]This is not rigorous if we are using a axiomatic set theory system, while we only use naive set theory to explain the notion.

```python
    Linear search algorithm to find the target element in the array.
    """
    for i in range(len(arr)):
        if arr[i] == target:
            return i  # Target found at index i
    return -1  # Target not found in the array

target = 5
target_list = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
print(linear_search(target_list, target))  # Output: 4
```

Problem 1.3.1 .
What is the time compelxity of the linear search algorithm in this call?

The answer is constant function, $f(n) = 1$, but not linear function, $f(n) = n$. The reason is that the length of the array is fixed here, instead of being a variable. Therefore, the time complexity of the linear search algorithm in this call is constant. The same goes for a given array of length 100,000, because the length of the array is fixed. This example may confuse some people, but noworries, we will explicately discuss the worst-case, best-case, and average-case time complexity in the following sections using linear search as an example.

§1.3.1. Worst-case Time Complexity (Big-O Notation)

As we said, when we evaluate the time complexity of an algorithm, we usually take certain considerations, and the most common one is how bad the algorithm can perform. This is called the worst-case time complexity, which provides an upper bound on the running time of the algorithm. We use big-O notation to represent the worst-case time complexity, for example, $O(n)$, $O(n^2)$, $O(\log n)$, etc.

Definition 1.3.1.1 (Big-O Notation).
By assuming the input of an algorithm is of variable size n, we use big-O notation to analyse the assymptotic upper bound of the time complexity when $n \to \infty$. We state that, in the worst case, the time complexity $T(n)$ of an algorithm with input size n has $T(n) = O(g(n))$, if

$$\exists (c > 0 \land r \in \mathbb{N}^{+}) \text{ such that } \forall n > r, 0 \leq f(n) \leq c \cdot g(n),$$

where $f(n)$ is the exact time complexity of the algorithm, and $g(n)$ is the upper bound of the time complexity.

In natural language, we say that the time complexity of the algorithm is $O(g(n))$ if there exists a constant c and a positive integer r such that, for all input sizes n greater than r, the time complexity of the algorithm is bounded above by $c \cdot g(n)$.

Still, we just use linear search as an example, though it's a quite special, because its actual worst-case time complexity is exactly the same as the big-O notation, while this does not always happen in other algorithms. We will see more examples in the future.

Example.
For the linear search algorithm, the worst-case time complexity is $O(n)$, where n is the length of the array. This is because, we have $f(n) = n$, and $g(n) = n$, when $c = 1$, $\forall n \in \mathbb{N}^+$,

$$0 \leq f(n) \leq 1 \cdot g(n).$$

In this case, we have more than what we need to know about the worst-case time complexity, because we need only $\forall n > r$, $0 \leq f(n) \leq c \cdot g(n)$, however we have $\forall n \in \mathbb{N}^+$, $0 \leq f(n) \leq c \cdot g(n)$. This is decided by the nature of the linear search algorithm, which is quite simple and straightforward. However, again, this is not always the case, and we will see more examples in the future.

§1.3.2. Best-case Time Complexity (Big-Omega Notation)

Opposite to the worst-case time complexity, we have the best-case time complexity, which provides a lower bound on the running time of the algorithm. We use big-Omega notation to represent the best-case time complexity, for example, $\Omega(n)$, $\Omega(n^2)$, $\Omega(\log n)$, etc.

Definition 1.3.2.1 (Big-Omega Notation).
By assuming the input of an algorithm is of variable size n, we use big-Omega notation to analyse the assymptotic lower bound of the time complexity when $n \to \infty$. We state that, in the best case, the time complexity $T(n)$ of an algorithm with input size n has $T(n) = \Omega(g(n))$, if

$$\exists (c > 0 \land r \in \mathbb{N}^+) \text{ such that } \forall n > r, f(n) \geq c \cdot g(n),$$

where $f(n)$ is the exact time complexity of the algorithm, and $g(n)$ is the upper bound of the time complexity.

In natural language, we say that the time complexity of the algorithm is $\Omega(g(n))$ if there exists a constant c and a positive integer r such that, for all input sizes n greater than r, the time complexity of the algorithm is bounded below by $c \cdot g(n)$.

Again, we use linear search as an example to illustrate the best-case time complexity.

Example.
For the linear search algorithm, the best-case time complexity is $\Omega(1)$, which means that the best-case time complexity is constant. This is because, we have $f(n) = 1$, and $g(n) = 1$, when $c = 1$, $\forall n \in \mathbb{N}^+$,

$$f(n) \geq 1 \cdot g(n).$$

§1.3.3. Average-case Time Complexity (Big-Theta Notation)

Definition 1.3.3.1 (Average Time Complexity).
Let $I(n)$ be the set of all input instances of size n, and let $T(n, x)$ denote the running time of an algorithm on input x in $I(n)$. Suppose there is a probability distribution $P(x)$ over these inputs for example, the uniform distribution, meaning that for all $x \in I(n)$ we have

$$P(x) = \frac{1}{|I(n)|}.$$

Then the average running time is defined as

$$T_{\text{avg}(n)} = \sum_{\{x \in I(n)\}} P(x) \cdot T(n, x).$$

In the case where every input is equally likely, this expression simplifies to

$$T_{\text{avg}(n)} = \frac{1}{|I(n)|} \sum_{\{x \in I(n)\}} T(n, x).$$

§1.4. Space Complexity

§2. Algorithm Complexity Rigorously Explained[6]

[6]This is optional, only for math lovers! Recommended for those who know function, set, and analysis well.